Designing a Query Language for the Semantic Web

Richard Fikes[1], Patrick Hayes[2], and Ian Horrocks[3]

[1]Knowledge Systems Laboratory
Computer Science Department
Stanford University
Stanford, California U.S.A.
1-650-725-3860
fikes@ksl.stanford.edu

[2]Institute for Human and Computer Cognition
University of West Florida
Pensacola, Florida U.S.A.
1-850-202-4416
phayes@coginst.uwf.edu

[3]Information Management Group
Department of Computer Science
University of Manchester
Manchester, M13 9PL, U.K.
44-161-275-6133
horrocks@cs.man.ac.uk

Abstract. We describe a formalism and protocol for communicating queries and answers between Semantic Web agents, as incorporated into the DAML+OIL Query Language proposal (DQL), and discuss some of the design issues which arise in query-answering in a Web context.

Introduction

In consultation with the DAML+OIL community, we recently designed DQL (DAML Query Language), a formal language and protocol intended for use between a querying agent (which we refer to as a *client*) and an answering agent (which we refer to as a *server*) when conducting a query-answering dialogue using knowledge represented in DAML+OIL [1]. The DQL Web site (http://www.daml.org/2002/08/dql/dql) provides links to current DQL implementations, including an XML Schema specifying a proposed external syntax for DQL and a DQL client with a Web browser user interface suitable for use by humans for asking queries of a DQL server [2]

This paper gives a general outline of DQL and discusses some of the design issues which have arisen in this effort.

DQL is intended to be a candidate standard language and protocol for query-answering dialogues among Semantic Web computational agents during which answering agents may derive answers. As such, it is designed to be suitable for a broad range of query-answering services and applications. Also, although DQL is specified for use with DAML+OIL, it is designed to be prototypical and easily adaptable to other declarative formal logic representation languages, including, in particular, the Ontology Web Language (OWL)[3] being developed by the W3C Web Ontology Working Group and the earlier W3C languages RDF[4] and RDFS[5].

Querying on the Semantic Web

The design of DQL is predicated on a number of basic assumptions about query-answering dialogs on the Web, and on the intended role of DQL.

First, there may be many kinds of server, with access to many types of information represented using many kinds of format. In particular, we cannot assume that all servers are likely to be accessible using conventional database querying protocols, although some may be. The design of DQL is intended to support relatively sophisticated query-answering services, but also to be useable by relatively simple services which offer a limited functionality but are easy to implement and can be deployed rapidly.

Second, and on the other hand, we can expect that servers using languages like DAML+OIL will be available which have non-trivial reasoning capabilities, perhaps themselves acting as intermediary agents between primary clients and more specialized servers.

Taken together, these suggest that the natural locus for a language like DQL is somewhat different from the conventional role of languages like SQL[6], which have been designed with conventional database technology in mind. Processes like consolidation, which in database querying are considered the task of the client, are likely in the SW to be incorporated into class reasoning (using, for example, spatial or temporal ontologies) performed by a server, which may then itself query a database system using SQL style queries. DQL supports query-answering dialogues in which the answering agent may use automated reasoning methods to derive answers to queries, as well as scenarios in which the knowledge to be used in answering a query may be in multiple knowledge bases on the Semantic Web, and/or where those knowledge bases are not specified by the querying agent.

Third, we must expect that some servers will have only partial information about the topic, some will have performance limitations and some will be simply unable to handle certain kinds of queries, so it is important that the querying protocol provide some means for the transfer of partial information about the query and about the querying process itself. These aspects of DQL have been chosen largely on pragmatic grounds. In this setting, the set of answers to a query may be of unpredictable size and may require an unpredictable amount of time to compute. DQL therefore provides an adaptable query answering protocol which both allows querying agents to specify the maximum number of answers required, and allows answering agents to return partial answers as they are computed.

Fourth, the set of notations and surface syntactic forms used on the Web is already large and various communities have different preferences, none of then universal: even within the nearest to a single established syntax, XML, there are many alternative ‘styles’ of notational design in use. The basic design issues in DQL seem to be independent of syntactic details, however, so rather than designing a particular surface syntax, we have tried to state the DQL specification at an ‘abstract’ or structural level, allowing essentially the same language to be implemented in different surface syntactic forms. We believe that this style of ‘meta-specification’ will be of more utility in a semantic web context than the more traditional approach, even though it provides only an incomplete specification of an actual language.

Finally, a basic premise of the SW is that the various notations used to express content on the Web will be united by a common semantic framework, and so the querying protocol should respect this framework and be described in terms of it as far as possible.

Query patterns and variables

Starting with the last of these, pieces of DAML+OIL markup (or of OIL, RDFS, OWL, etc.) are thought of as logical assertions. It is natural therefore to think of a query as something like a goal to be proved, and to think of using bindings to variables in the query as answers to the query. This conventional picture squares well with the SW semantic foundation and is consistent with the Codd database model and many other applications, so we have adopted it as the core of DQL.

In order to make it precise, however, it requires that we take the language in question – in our case, DAML+OIL – and either identify those parts of its syntax which are to be classed as ‘variables’, or else to extend the language by adding such items to its syntax; and having done so, to specify what counts as a binding. In fact, DAML+OIL has no suitable notion of a variable, so the central item in a DQL query is a query pattern consisting of a DAML+OIL KB modified so that some of its ‘names’ (literals or URI references) are replaced by variables, which are therefore required to be syntactically distinguishable from DAML+OIL expressions themselves. Since the DQL query pattern is not itself legal DAML+OIL, the task of the server is to search for legal DAML+OIL expressions which can be formed by substituting a DAML+OIL name for the variables in the pattern and which are entailed by its knowledge base. (There is a delicate point here, in fact, concerning what exactly counts as ‘substituting a DAML+OIL name’. We will return to this later.) The use of ‘entailment’ here is what most clearly distinguishes DQL from SQL, since although of course a database may be understood to entail its table entries considered as atomic assertions, entailment also allows more complex relationships to hold which may be much more expensive to compute.

The essential semantic connection between the query and the server’s knowledge base is thus a composition of two parts: entailment according the SW semantic conventions, and instantiation of pattern variables. Intuitively, the query is asking the server: what bindings to these variables do you know which would make this true? It is important to note that this is not exactly the same as asking, what things would make this true? Logical reasoners may be capable of proving that something exists which would satisfy the query, without having a term to bind to the variable. (For example, if the query asks about Joe’s siblings, the KB may know that the cardinality of the class restriction of siblingOf to Joe is one.) Under these conditions, a server may be unable to answer the query as posed. We will return to this issue later.

Answers and bindings

Specifying the pattern does not indicate how the answers – the bindings to the pattern variables – are to be returned from the server to the client. DQL allows this to be done by using an *answer pattern*, a suitable instantiation of which constitutes an answer to the query. If no answer pattern is specified, the query pattern is used. The answer pattern can be any expression, so that one option is simply to list the variables in a particular order, producing an ‘answer vector’. Different forms of the answer pattern can be used to provide a limited amount of control over the formatting of query answers.

Not all the pattern variables need be instantiated in an answer, which allows the server to respond with partial information when given a complex query. We have to specify the semantics of this. Missing variables in an answer are understood as existential claims by the server. Thus for example the query pattern (expressed using N-triples notation)

```
?p Ex:owns ?c .

?c rdf:type ex:Car .

?c ex:hasColor “Red” .
```

might produce the answer binding ?p = ex:Joe, with no binding for ?c. This would be understood to mean that Joe has *a* red car, without identifying the particular car. (Note, this answer does not mean that no binding is available, or that the name of the car is unknown to the server: further queries may be able to elicit a more complete answer.) In order to allow the client some control over this, a pattern variable may be indicated in a query as a *must-bind variable*, in which case the server is required to provide a binding; so if it is unable to compute a binding for such a variable, it is required not to answer.

Given this semantics for unbound variables in answers, it is natural to think of pattern variables as playing the role of an existential variable in a query. DQL capitalizes on this analogy by allowing the client to include variables which are not intended to be bound in an answer, called *don't-bind variables*. These can be used to increase the expressive power of the query pattern in some cases. Thus, the above query with ?c as a don't-bind variable would be a request for the names of owners of red cars. All other pattern variables are called may-bind variables, which is the default. Any DQL query syntax must somehow distinguish the three kinds variable in each query.

There are several other components to a DQL query which will be described later.

Answer delivery

A query may have any number of answers, including none. In general, we cannot expect that all the answers will be produced at once, or that the client is willing to wait for an exhaustive search to be completed. We also cannot expect that all servers will guarantee to provide all answers to a query, or to not provide any redundant or duplicate answers. DQL attempts to provide a basic kit of tools to allow clients and servers to interact under these conditions. The basic presumption underlying this design is that communication between client and server is likely to be conducted using Internet data transfer protocols.

We allow answers to be delivered by the server in *bundles*, but for the *size* of the bundle to be restricted above by the client. Part of a query therefore is a bundle size bound. (If omitted, the size bound is set to infinity: this could be suitable for a conventional database querying application; a the size bound of zero can be treated by a server as a termination of the query answering process.) The server is required to deliver an *answer bundle* containing at most this many answers in response to the query. The answer bundle must also contain a special item which is either a *server continuation* or one or more character strings called *termination tokens*. A termination token indicates that the server will not deliver any more answers to the query, and a server continuation represents a commitment by the server to deliver another answer bundle. Passing the continuation back from the client to the server constitutes a request for the next answer bundle to be delivered from the server to the process which passes the continuation.

This description is deliberately somewhat vague as to the exact nature of continuations. Different servers may use this in different ways. Some DB servers may generate a complete table of answers, store it in association with a record of the query, and then use an index or hash code keyed to the query reference as a continuation. Others may take advantage of the protocol to force the client to store enough information to enable them reconstruct the state of a search process. Still others may use a very simple tag, or ignore the continuation altogether, if the state of the interaction is maintained by other protocols. Note that the delivery of a continuation is not a commitment to provide any more answers. If for example the server is unable to reconstruct the state of a query process, it is always safe to return an answer bundle containing the termination token “End”.

Note that a client can pass the continuation to some other process or client which will receive the next answer bundle from the server, providing a limited degree of transaction-handling between systems of clients dealing with query answers. A continuation may also allow the client to change the bundle size for the next bundle.

DQL recognizes three termination tokens. “End” simply indicates that the server is unable to deliver any more answers; it is conventionally used to terminate the process of responding to a query. One possible response to any query is a single bundle containing “End”, indicating that there are no answers. “None” expresses an assertion by the server that no other answers are possible. This is a definite assertion, which should be used with care, particularly when in response to a query containing a don’t-bind variable, where is has the semantic force of a negated existential, i.e. a universal negation. Finally, the termination token “Rejected” can be used by a server to indicate that the query is outside its scope for some reason, e.g. by being posed in a subset of the language which it is unable to process, or by being in some way ill-formed. This is a crude device for expressing what could be a complex topic, but servers may also define their own termination tokens to be used in conjunction with the DQL tokens, which can be used to express more nuanced forms of rejection.

The use of “None” would be appropriate in a case where a server has access to a collection of data which is known to be complete or exhaustive in some way, such as a database of employees of a company. Suppose a query asks for all employees with a salary over $200K, and the returned answer bundle is empty, terminated with “None”. This would be sufficient grounds for the client to conclude that the company has no employees with that salary. Notice that the termination token “End” would *not* provide this kind of a guarantee, given the monotonic semantics of DAML+OIL. To treat an “End” token as though it meant “None” would be to make a ‘closed-world assumption’, which is not strictly valid. The distinction between these tokens was motivated in part by the widely noted utility of closed-world reasoning. Making the distinction explicit in the exchange protocol provides a way to express closure without forcing clients to draw invalid conclusions in cases where a closed-world assumption is inappropriate.

These conventions, taken together, allow a simple expression of a 'yes/no' query. This can be expressed by a query pattern with no variables; an answer bundle containing an empty answer indicates that the pattern is entailed by the KB; an empty answer bundle containing the termination token "None" indicates that it is known to not be entailed; and any other empty answer bundle indicates that entailment cannot be determined.

Other features

Several other features have been added to the DQL query specification, in response to user requests. We have attempted to make these optional wherever they could impose a burden on implementers of simple servers.

Asking about the source of information.

In a Web setting, we can expect that there will be servers which will access several – perhaps many - knowledge bases in an attempt to respond to a query. It may be, however, that a client wishes to limit its query only to a limited set of sources, or at any rate wishes to know what KBs were used to generate the reply. This could get arbitrarily complex, if one considers possibilities such as having meta-KBs which describe other KBs, and allowing queries about the nature and kinds of authority which a KB can be said to have. While we anticipate that such issues will eventually arise, DQL has a simple mechanism which provides for a basic degree of expressiveness by taking advantage of the conventions described previously. A query includes an answer KB pattern, which is either a variable or a list of URI references denoting KBs. The server is required to only use sources which are listed in the binding to this (very simple) pattern. By either providing a list, or by inserting a must-bind variable, the client can restrict, or request information concerning, the sources used by the server. Note that a server may consider itself to be a KB, even if it uses information from many other sources; it would then be up to a client to decide whether this KB was trustworthy or otherwise acceptable.

Asking about the number of answers.

Many database servers record information about the numbers of entries in their datatables, and can rapidly respond to requests for this information, which can then be used by a client to optimize subsequent queries. Such information may be much harder, or impossible, to compute for other KBs, however; and for many KB's based on logic, information about the number of answers often cannot be said to be entailed by the KB itself, so cannot be integrated into the query pattern syntax without breaking the underlying semantic model. Thus, information about the number of answers is not always available, and needs to be handled specially, but can be useful.

After extensive discussions, we have included this as an optional part of a query. DQL allows a query to include an answer number request, which is a variable. The binding to an answer number request must be a numeral.

If it is a must-bind variable and the server is unable to provide the information, then no answers can be delivered, so the safest client option is to make it a may-bind variable. Note that the answer number here is the number of answers which will be delivered, which need not be the number of tuples of entities which satisfy the query. There may be other entities unknown to the server, and several of the answers may indicate the same entities, perhaps under different names. (If the server delivers the termination token "None" then the number of answers is an upper bound.)

It is important to note that in general, the number of answers need not be the same as, or even simply related to, the number of distinct entities which satisfy the query. It is possible for the set of answers to contain redundant answers, for example, and it is possible that there are entities which have no associated term which can be provided as an answer to the query. Unlike conventional databases, sources of knowledge on the semantic web cannot be assumed to be complete and exhaustive, even concerning the items they provide information about.

The situation is rendered more complicated by the ability of languages like DAML+OIL and OWL to express information about cardinalities. For example, it is possible to define a class of people who have exactly three brothers, and to assert that Joe is in that class. A query about Joe's brother's expressed using the property of brotherhood with a variable as the value would not thereby produce three answers, however, since this cardinality information would not in itself supply three bindings for the variable in such a query. In fact, it is possible that no answers would be available in this case to such a query, even though an explicit query about the cardinality of the class of Joe's brothers might produce the answer `xsd:integer"3".` The ability of the server to respond with a suitable answer depends on the query being formulated so that the server is able to provide a binding to a query variable.

Asking under assumptions

It is often convenient for the query to supply a temporary assumption to be used as one of the KBs in responding to the query, for example when a query follows from an earlier query which has established the truth of the assumption. DQL allows a query to specify such a 'query premise' either explicitly or by giving a URI reference to DAML+OIL ontology. The KB specified in the query premise is considered to be included in the answer KB. The query premise is optional: omitting the query premise is equivalent to supplying an empty query premise.

DQL does not allow query premises to be patterns containing query variables. Allowing this would impose a burden on DAML+OIL reasoning engines which would go beyond the DAML+OIL specification, since the only natural interpretation of such variables would be that they were universally quantified across the KB and the query.

What counts as a binding?

The entire discussion has been predicated on the assumption that there is a clear notion of what it means to bind an expression to a variable. This is not entirely clear, however, from just the description of DAML+OIL given in its specifications. Two issues in particular have given rise to discussion, one arising from the use of RDF to encode DAML+OIL syntax, and the other concerned with the creation of new names by the server.

RDF lists and DAML+OIL expressions.

DAML+OIL is typical of the emerging group of W3C standard ontology languages (others include RDFS and OWL) in having a syntax which is formally described as a set of RDF triples. Since RDF does not include any syntactic provision for describing anything more complicated than a triple, more complex DAML+OIL expressions are encoded into RDF by thinking of them as list constructions and using an auxiliary RDF vocabulary to describe the resulting lists. The relevance of this for DQL comes in the cases where such a list could be the appropriate binding of a query variable. If we interpret the DAML/RDF syntax strictly, the binding for such a variable would be a ‘node’ in the RDF graph syntax which is being used to encode the list, rather than the list of items itself: in fact, strictly speaking, the list is described by the RDF syntax, rather than being a syntactic part of it, and hence an appropriate binding to a variable. Put another way, the names in DAML/RDF are not always the most natural names in DAML+OIL, considered as a language in its own right. On the other hand, the natural interpretation of DAML+OIL would suggest that the list (that is, the sequence of items in the list, not the list considered as a data-structure) is best considered a syntactic entity and hence an appropriate – and certainly more informative – answer to the query. (Similarly for OWL: the issue does not arise for RDFS, which does not use these syntactic encodings.)

In cases like this, therefore, there is a systematic ambiguity between understanding the language to be DAML+OIL (or OWL) considered as a language with an abstract syntax, or to be the RDF encoding of DAML+OIL (or OWL). While this is strictly speaking a matter for the relevant language specs to decide, the use of a query protocol based on syntactic matching in this way does throw the issue into a somewhat sharper relief.

We do not yet have a fully satisfactory response to this issue. Currently the DQL specification leaves the precise definition of 'binding' to the server; it would be more satisfactory, however, to have a general solution for this problem, which seems likely to arise frequently. Note that if a server were to only follow the strict RDF syntax, it would be possible for a client to elicit the required information, if rather awkwardly, by a succession of queries using the RDF list (ie 'collection') vocabulary explicitly to discover all the items in the list one by one. So the distinction can be phrased as the question of whether it is the client or the server which has the responsibility, in a case like this, to understand the way that RDF can be 'parsed' as DAML+OIL. If the server will not do it, the client must. Similar considerations apply to queries made using OWL.

Reporting existentials.

All of the SW ontology languages support some form of existential assertions and reasoning. That is, they support assertions to the effect that things exist, without necessarily providing names for them. In the RDF syntax model, the relevant existential variables are called 'blank nodes', and are understood to be restricted in scope to the KB in which they occur. The question then arises, what answer can be given to a query when the relevant binding is a blank node, i.e. an expression which is meaningless outside of the KB?

There are various options here, ranging from simply declaring that no answer can be given in cases like this, to providing a special DQL format for delivering such 'existential' answers. After considerable discussion, however, DQL essentially does nothing, and we assume that servers will be able to create new URI references to designate such entities, which can then be provided as answer bindings and used in subsequent queries. This represents a non-trivial requirement on a server designed to usefully interact using DQL, however. Note that according to our criteria of entailment, it requires the server to consider that its KB actually entails the 'new' instance of the answer pattern, even though in this case, we are assuming, it is was in fact only able to prove an existential assertion. In effect, the server is required to perform an action of Skolemization, in order to create a name to use as an answer binding. Although Skolemization is not strictly logically valid, we feel that the W3C protocols surrounding creation and ownership of URIs are sufficiently clear to ensure that no confusion will arise from this. Even in the case where a client may be unable to distinguish such a Skolemized URI reference from any other URI reference, any attempt to use it in another query to any other server will be referred back to this server, and no other server will have any information about it.

DQL does not require that the server provide such Skolemized names in response to a query; in general, the DQL protocols allow servers to be arbitrarily incomplete.

As illustrated by the example of Joe's brothers, it is possible for a knowledge base to entail an existential assertion without making it directly. In this case, to require that a suitable name be generated in order to supply an answer would be a very heavy burden on the design of a server. Very few, if any, existing reasoning engines would be capable of generating a suitable answer in all such cases.

Organizing the set of answers

The set of all answers sent to the client by the server in a query-answering dialogue is called the *response collection* of that dialogue. While there are no global requirements on a response collection other than that all its members are correct answers, clients may find it useful to know whether a given server ensures that its response collections contain no duplicate or redundant answers. For some servers, this would be a potentially expensive requirement, and imposing it as part of an intended standard would impose a high initial implementation cost for simple servers. On the other hand, a server which is able to deliver non-redundant responses may wish to advertise this useful quality. We have therefore provisionally defined a series of conformance levels which refer to this issue.

A server which always produces a response collection which contains no overt duplications can be called *non-repeating*. However, there are more subtle kinds of redundancy. Say that an answer subsumes another if the second is an instance of the first, ie if one can get to the second from the first by binding to more variables. If the query language itself can express universal quantification, this relationship can get quite complicated, since a binding to one variable can then include further variables. A server which always produces a response collection in which no answer is subsumed by any other answer in the set is called *terse*. Finally, a server which guarantees that its response collection will always be correctly terminated by "None" can be called *complete*. Complete terse servers offer a very strong guarantee about the quality of their responses.

Note that if a terse server sends a client an answer with an unbound may-bind variable (i.e., the answer does not provide a binding for that variable), the server cannot later send an answer that contains the same bindings as those in with the addition of a binding for the unbound variable, since it would be subsumed the first answer

Currently, DQL has no provision for requesting that answers be supplied in a particular order, or that the answer collection be grouped or organized in particular ways. Although such provisions could be added, they would complicate the DQL query syntax, and are not a natural fit for the underlying semantic model. Also, it seems likely that servers which are able to deal with such requests will be SQL servers in any case, and DQL is not intended to replace or subsume SQL.

Utilizing the expressive power of DAML+OIL

Query languages for DLs, and other logic-based KR formalisms, often include explicit "structural queries", i.e., queries asking about the subsumers, subsumees, instances, etc. [7,8,9]. In DQL, these kinds of question can be formulated using the standard query mechanism, taking advantage of the expressive power of the DAML+OIL language itself. For these cases, the RDFS subset of DAML+OIL is sufficient.

For example, the answers to the query using the query pattern

```
?x rdf:subClassOf ex:Person .
```

where ?x is a must-bind variable will be all the known subclasses of Person. Similarly, obtaining the set of individuals that instantiate Person can be achieved using the query pattern

```
?x rdf:type ex:Person .
```

where ?x is again a must-bind variable. This ability is limited to concepts which can be expressed using DAML+OIL, of course: for example, there is no way in DAML+OIL to express the concept of a most general subclass or a most specific type. Nevertheless, it seems more appropriate for the querying system to restrict its expressive abilities to those of the content language used in the knowledge bases being queried, since extending this expressiveness is liable to impose greater computational burdens on a server that are defined by the specification of the language it uses.

Some SQL-style queries can be expressed using a similar technique. For example, a simple relational table might be encoded in DAML+OIL as a collection of assertions using `rdf:value`. with the following format:

```
ex:Joe rdf:value _:x

_:x rdf:type ex:employeeInfo

_:x ex:surname "Jones"

_:x ex:SSnumber "234-55-6789"

_:x ex;age xsd:number^^"43"

_:x ex:location ex:marketing
```

where the value of rdf:type is the 'table entry' and the table name is its type. The SQL command
'*select SSnumber from employeeInfo*' then translates into the DQL query pattern

```
?x rdf:type ex:emplyeeInfo

?x ex:SSnumber ?y
```

with `?y` a must-bind and `?x` a don't bind variable. More complex SQL conditions can be expressed by more complex DAML+OIL patterns – for example, a conditional selection can be expressed as a DAML+OIL restriction on a class such as `ex:employeeInfo` - but these will provide answers only if the server is able to perform the appropriate reasoning. In general, in contrast to the presumptions of the SQL querying model, the DQL assumption is that nontrivial inferences about the data are performed by the server rather than by the client.

We envision that there will be semantic web query servers which will accept input in DQL, process the query appropriately, and then themselves act as clients in further DQL queries and SQL transactions with conventional databases, in order to determine appropriate bindings for the DQL answering protocols. In this way, answers can be extracted by DAML+OIL inference processes in ways which are inaccessible to conventional database technology.

Some design lessons learned

In many ways the design of DQL represents a series of compromises between potentially conflicting requirements. Users of a querying protocol tend to want more features in queries, particularly those which enable them to rapidly and easily extract information from certain kinds of data source, and to desire semantically precise, informative and complete answer responses. The nature of the sources, and the kinds of information which it is reasonable to extract from them, vary among user constituencies, however. It seems clear that a language which would fully satisfy all potential users or client designers would be complex, unwieldy and burdensome to support as a single standard. We have attempted to avoid 'feature-bloat' in stating queries, but a DQL query can already have as many as seven items (query pattern, answer pattern, KB pattern, answer number pattern, list of must-bind variables, list of no-bind variables, bundle size bound) and others may be needed in order to support network protocols (identity of the client, for example). When expressed in an XML syntax, the resulting query can be somewhat overwhelming.

Seen from the perspective of the server, increased complexity of a query, and particularly increased precision and completeness in the responses, represents more work to do. The logical expressiveness of Web formalisms such as DAML+OIL and OWL is already somewhere beyond the 'sweet spot' on the expressiveness/intractability spectrum: the computational complexity of OWL reasoning is at the limits of current technology for a complete, effective reasoner; to require in addition that the server must provide bindings for all the entities which could be inferred to exist would take the task outside the range of currently available technology. Clearly such a requirement would not be feasible; and in any case, part of the semantic web community envisions the deployment of incomplete but useful reasoners which can handle only a small part of the full power of these languages.

This is why the DQL specification tries to avoid imposing completeness conditions on servers, in a notable contrast to database querying specifications.

We believe that this process of designing DQL may be representative of an entirely new set of design constraints which will be typical on the semantic web. We began with a simple, well-tried notion taken from an existing field – the idea of answers as bindings to variables made during a proof of an entailment, which has long been a staple of AI and logic programming – and found that to deploy it successfully in a wider context it was necessary to adapt and modify it to conform to the requirements arising from a much wider user community. Consulting a wider community of potential users in this way, however, produces a constant pressure on the designers towards 'feature-bloat', which must be resisted, even at the cost of some user dissatisfaction. Like other standards being developed for Web use, it is vital that the specification not rule out very simple, 'incomplete' implementations, while also providing for more advanced implementations, especially existing implementations already in large-scale commercial or industrial use. Typically, it is necessary to provide multiple levels of conformance. It was necessary to understand how to leverage new abilities provided by the use of other semantic web standards; and finally, we found that it was necessary to identify and to clarify as far as possible the distinction in intended *uses* between this proposal and existing query languages, if only to clarify why it is not more like them, does not subsume them, or fails to provide features which some user communities have come to think of as 'standard'.

Acknowledgements

This research was supported by the Defense Advanced Research Projects Agency under contracts F30602-00-0579-P00001 (Fikes, Horrocks) and F30602-00-2-0577 (Hayes). We would also like to acknowledge the valuable contributions of Yulin Li and the other members of the Joint United States / European Union ad hoc Agent Markup Language Committee, especially Peter Patel-Schneider.

References

1. DAML+OIL http://www.daml.org/language/.
2. R. Fikes, P. Hayes, and I. Horrocks; "DAML Query Language (DQL) Abstract Specification"; http://www.daml.org/2002/08/dql/dql.
3. M. Dean, et al; "Web Ontology Language (OWL) Reference Version 1.0"; July 29, 2002; http://www.w3.org/TR/owl-ref/
4. G. Klyne and J. J. Carroll; Resource Description Framework (RDF):Concpets and abstract Syntax, W3C working draft, http://www.w3.org/TR/rdf-concepts/
5. D. Brickley and R.V.Guha; RDF Vocabulary Description Language 1.0: RDF Schema. W3C working draft, http://www.w3.org/TR/rdf-schema/

6. "Information Technology --- Database Languages --- SQL"; ISO/IEC 9075:1992; American National Standards Institute; New York, N.Y.
7. G. Karvounarakis; “The RDF Query Language (RQL)”; Institute of Computer Science, Foundation of Research Technology; Hellas, Greece; http://139.91.183.30:9090/RDF/RQL/.
8. Franz Baader, Hans-Jürgen Bürkert, Jochen Heinsohn, Bernhard Hollunder, Jürgen, Müller, Bernard Nebel, Werner Nutt and Hans-Jürgen Profitlich, “Terminological Knowledge Representation: A Proposal for a Terminological Logic”, Technical Memo, DFKI, 1991.
9. Sean Bechhofer, Ian Horrocks, Peter F. Patel-Schneider and Sergio Tessaris, “A Proposal for a Description Logic Interface”, in Proc. of DL’99, pp. 33-36, (1999).